Computer Science Department

TECHNICAL REPORT

A FORMAL NOTION OF PROGRAM-BASED
TEST DATA ADEQUACY

BY

MARTIN D. DAVIS

AND

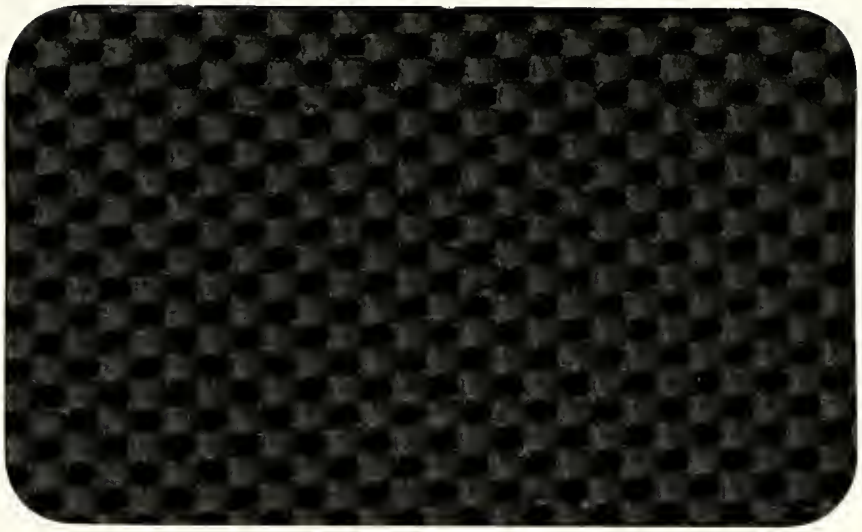
ELAINE J. WEYUKER

TECHNICAL REPORT #50

OCTOBER, 1982

NEW YORK UNIVERSITY

Department of Computer Science
Courant Institute of Mathematical Sciences
251 MERCER STREET, NEW YORK, N.Y. 10012



A FORMAL NOTION OF PROGRAM-BASED TEST DATA ADEQUACY

BY

MARTIN D. DAVIS

AND

ELAINE J. WEYUKER

TECHNICAL REPORT #50

OCTOBER, 1982

This research was supported in part by the National Science Foundation under grants MCS-80-02438 and MCS-82-01167.

A FORMAL NOTION OF PROGRAM-BASED TEST DATA ADEQUACY

Martin D. Davis and Elaine J. Weyuker

Introduction

We propose a definition of the notion of adequacy of test data and discuss justification, difficulties, and properties of the notion. It is not the purpose of this paper to suggest a definite practically applicable criterion of test data adequacy. Rather we present a theoretical analysis which, it is believed, gives insight into such questions as:

a) For a given program, what points must belong to a test set in order that it may be deemed adequate ?

b) For a given program, how many points must belong to an adequate test set ?

c) What kind of approximation to "correctness" can be provided by the knowledge that a program has been "adequately" tested ?

We believe, in general, that an adequacy criterion should be invoked only after the test data fails to expose errors. Clearly, as long as there is an element of the test set on which the program does not agree with the specification, we know that the test data is still doing its job and that testing (and subsequent debugging) must continue. (In this paper, we ignore the question of whether and how we can tell whether a program agrees with a specification at a particular point. However, see [15], [3].) Once the program does agree with the specification on all elements of a set of test data, we must decide whether the testing phase can end, and hence we will need to invoke some kind of adequacy criterion.

This research was supported in part by the National Science Foundation under grants MCS-80-02438 and MCS-82-01167.

For a given program P, we write P(c)=b to mean that the program P on input c halts with output b. We write P ≡ Q (P is equivalent to Q), where P and Q are programs, to mean that P(c) = Q(c) for every input c. In particular this implies that for each c, P(c) is defined if and only if Q(c) is defined.

Our analysis will deal only with adequacy criteria which are entirely program-based in the sense that they depend completely on the program as written. Since no set of test data can distinguish two programs with identical input-output behavior, the broadest conceivable program-based adequacy criterion would be to regard a set of test data as adequate for a given program if the input-output behavior of the program on the test set is different from that of all programs that are inequivalent to it. However, it is easy to see that no set of test data can hope to be adequate in this sense unless it contains every point in the domain of the program. For, if the point c on which program P is defined is omitted from a test set, then we can easily construct a new program Q whose input-output behavior is identical to that of P for all elements of the domain of P other than c, but such that P(c)≠Q(c).

We shall develop and study a notion of test data adequacy which approximates the above notion, while avoiding the "diagonal" construction by which Q was obtained from P.

The Programming Language

We now define our programming language. Although most of the results of the paper are not really dependent on the particular details of this language, it is necessary to have an explicit syntax.

Our language will contain a finite number of identifiers whose range is the integers (positive, negative, or zero). The language also contains a finite number of constants representing particular integers; we will assume that all numbers we encounter as input or output values are represented by corresponding

For a given program P, we write P(c)=b to mean that the program P on input c halts with output b. We write P ≡ Q (P is equivalent to Q), where P and Q are programs, to mean that P(c) = Q(c) for every input c. In particular this implies that for each c, P(c) is defined if and only if Q(c) is defined.

Our analysis will deal only with adequacy criteria which are entirely program-based in the sense that they depend completely on the program as written. Since no set of test data can distinguish two programs with identical input-output behavior, the broadest conceivable program-based adequacy criterion would be to regard a set of test data as adequate for a given program if the input-output behavior of the program on the test set is different from that of all programs that are inequivalent to it. However, it is easy to see that no set of test data can hope to be adequate in this sense unless it contains every point in the domain of the program. For, if the point c on which program P is defined is omitted from a test set, then we can easily construct a new program Q whose input-output behavior is identical to that of P for all elements of the domain of P other than c, but such that P(c)≠Q(c).

We shall develop and study a notion of test data adequacy which approximates the above notion, while avoiding the "diagonal" construction by which Q was obtained from P.

The Programming Language

We now define our programming language. Although most of the results of the paper are not really dependent on the particular details of this language, it is necessary to have an explicit syntax.

Our language will contain a finite number of identifiers whose range is the integers (positive, negative, or zero). The language also contains a finite number of constants representing particular integers; we will assume that all numbers we encounter as input or output values are represented by corresponding

constants of our language. Arithmetic expressions are constructed using constants, identifiers, and arithmetic operators in the usual manner. An assignment statement is one of the form:

VAR ← EXP

where VAR is an identifier and EXP is an arithmetic expression. A continue statement, which will be written

continue

is a dummy statement, like the "continue" statement of Fortran, which we take to be simply an assignment statement of the form:

VAR ← VAR

where the same identifier occurs on the left as on the right.

A predicate is a condition having one of the forms:

EXP1=EXP2, EXP1≠EXP2, EXP1<EXP2, EXP1≤EXP2,

where EXP1 and EXP2 are arithmetic expressions. The notion of program body is defined recursively as follows:

1. An assignment statement is a program body.

2. if PRED then P
   else Q

is a program body provided that PRED is a predicate and P and Q are program bodies.

3. while PRED do P

is a program body if PRED is a predicate and P is a program body.

4. P
   Q

is a program body if P and Q are program bodies.

An input statement has the form:

input VAR

and an output statement has the form:

output VAR

where VAR is an identifier. Finally a program consists of an input statement followed by a program body followed by an output statement. When no confusion results, we sometimes use the same symbol to represent a given program and the program body from which it is formed. Since our language consists of entirely familiar locutions, there is no need for us to specify its formal semantics.

Our study of test data adequacy will assume a notion of size of a program and we shall write $|P|$ for the size of program P. The question of how to measure the size or complexity of a program is a difficult one which many people have considered [2, 5, 6, 9, 10, 17]. We shall find it most useful to define $|P|$ to be the maximum of two quantities associated with the program P, namely:

(1) The number of arithmetic operations in P plus the number of ←'s. (Note that since the continue statement is an abbreviation for a statement of the form VAR ← VAR, each such statement adds one to this count.)

(2) The number of occurrences of predicates in P.

(We also compute $|R|$, where R is a program body, in the same way.) Note that with this definition, for each positive integer n, there are only finitely many programs P such that $|P| < n$.

Size-adequacy

We will call a test set T size-adequate for a program P if for each program P′ which is not equivalent to P but for which $P'(t) = P(t)$ for each $t \in T$, we have $|P'| > |P|$.

Thus, our first suggested approximation to the ideal adequacy notion is to ask for a set of test data which serves to distinguish P, not from all programs which are inequivalent to P, but merely from those inequivalent programs whose size is no larger than that of P. The notion of size-adequacy has a number of interesting properties. As our discussion below will demonstrate, it subsumes

output VAR

where VAR is an identifier. Finally a program consists of an input statement followed by a program body followed by an output statement. When no confusion results, we sometimes use the same symbol to represent a given program and the program body from which it is formed. Since our language consists of entirely familiar locutions, there is no need for us to specify its formal semantics.

Our study of test data adequacy will assume a notion of size of a program and we shall write $|P|$ for the size of program P. The question of how to measure the size or complexity of a program is a difficult one which many people have considered [2, 5, 6, 9, 10, 17]. We shall find it most useful to define $|P|$ to be the maximum of two quantities associated with the program P, namely:

(1) The number of arithmetic operations in P plus the number of ←'s. (Note that since the continue statement is an abbreviation for a statement of the form VAR ← VAR, each such statement adds one to this count.)

(2) The number of occurrences of predicates in P.

(We also compute $|R|$, where R is a program body, in the same way.) Note that with this definition, for each positive integer n, there are only finitely many programs P such that $|P| < n$.

Size-adequacy

We will call a test set T size-adequate for a program P if for each program P′ which is not equivalent to P but for which $P'(t) = P(t)$ for each $t \in T$, we have $|P'| > |P|$.

Thus, our first suggested approximation to the ideal adequacy notion is to ask for a set of test data which serves to distinguish P, not from all programs which are inequivalent to P, but merely from those inequivalent programs whose size is no larger than that of P. The notion of size-adequacy has a number of interesting properties. As our discussion below will demonstrate, it subsumes

such well-known adequacy criteria as branch coverage and mutation analysis. Unfortunately, we have the following result:

THEOREM 1. Suppose that the program P is non-minimal, in the sense that there is a program Q such that $Q \equiv P$ and $|Q| < |P|$. Then no test set T can be size-adequate for P unless T includes all elements of the domain of P.

Proof: Let P(c) be defined, where c does not belong to T. Let the program Q have the form:

```
input x
   R
output y
```

Let b be a constant which represents a number different from P(c), and let V be the program:

```
input x
if x=c then y ← b
       else R
output y
```

Since each of the pair of numbers whose maximum is $|P|$ is increased by 1 in forming V, we have $|V| = 1+|Q| < |P|$. Moreover, since $V(c)=b \neq P(c)$, V is not equivalent to P. Hence T is not size-adequate. •

Thus, the only programs which can possess size-adequate test sets are those programs which are optimal in the sense of being of minimum length among all equivalent programs. This is clearly unacceptable, since we obviously cannot in general count on writing minimum length programs, and we must be prepared to test any program and decide in a coherent fashion whether or not the testing process is complete. The difficulty arises from the possibility of constructing programs like V in which an equivalent of P is "embedded," thus permitting a diagonal

construction. This kind of embedding can occur in rather subtle ways. For example, suppose a program Q equivalent to P with $|Q|<|P|$ has the form:

```
input x
u ← EXP
   R
output y
```

where the identifier x does not occur in the program body R, and EXP is some arithmetic expression in x. With c and b as above, let the program W be:

```
input x
u ← EXP
if x = c then y ← b
         else R
output y
```

Then $|W|<|P|$ since $|W|=1+|Q|<1+|P|$, and $W(c)\neq P(c)$.

In order to proceed it is clearly necessary to modify the definition of size-adequacy in such a way that an adequate set of test data is not expected to distinguish a program P from programs like V and W in which P is "embedded." The problem is to state precisely when one program is to be regarded as embedded in another. For this purpose we begin with three reduction rules:

1. Replace some assignment statement by continue
2. Replace an if statement:

```
if PRED then P
        else Q
```

   by P.

3. Replace an if statement:

```
if PRED then P
        else Q
```

   by Q.

construction. This kind of embedding can occur in rather subtle ways. For example, suppose a program Q equivalent to P with $|Q|<|P|$ has the form:

```
input x
u ← EXP
   R
output y
```

where the identifier x does not occur in the program body R, and EXP is some arithmetic expression in x. With c and b as above, let the program W be:

```
input x
u ← EXP
if x = c then y ← b
         else R
output y
```

Then $|W|\leq|P|$ since $|W|=1+|Q|<1+|P|$, and $W(c)\neq P(c)$.

In order to proceed it is clearly necessary to modify the definition of size-adequacy in such a way that an adequate set of test data is not expected to distinguish a program P from programs like V and W in which P is "embedded." The problem is to state precisely when one program is to be regarded as embedded in another. For this purpose we begin with three reduction rules:

1. Replace some assignment statement by continue
2. Replace an if statement:

```
if PRED then P
        else Q
```

   by P.

3. Replace an if statement:

```
if PRED then P
        else Q
```

   by Q.

We say that M reduces to N if the program N can be obtained from M by applying these reduction rules. Finally, we say that M is embedded in N if N reduces to some program which is equivalent to M. Note that in the examples above P is embedded in both V and W.

A Modified Notion of Size-adequacy

In what follows, we will call a finite test set T adequate for a program P if for each program P′ such that P is not embedded in P′, but for which P′(t) = P(t) for each t ε T, we have |P′| > |P|. It is this notion of test data adequacy that we now proceed to investigate.

The negative result embodied in Theorem 1 indicates the pervasiveness of diagonal constructions made possible by self-reference. It is just this phenomenon which leads to unsolvability results in computability theory, Gödel undecidability in logic, and paradoxes in set theory. Its occurrence here is not a mere artifact of our approach, but shows once again the impact of these highly abstract and theoretical considerations on efforts to guarantee the quality of software. Our solution to this dilemma does have an unfortunate ad hoc character. But nevertheless as we have just seen, it does succeed in capturing the idea of distinguishing a given non-pathological program from a very large class of programs inequivalent to it.

Formally, our results would remain unchanged if we computed the size of a program as simply the total number of occurrences of predicates in the program, or the closely related cyclomatic number [5,6,9,10]. However, if we had made this choice, it would have been possible to construct programs of size 1 with any desired input-output behavior on any finite set of points (for example, by using interpolating polynomials); this would render our definition trivial.

As we have noted, our concern in this paper is exclusively with program-based adequacy criteria. Now in practice, an adequacy criterion will not

be invoked unless it is believed that the program agrees with its specification on the test set. Hence, it might be thought appropriate to include correctness on the test set as part of the definition of adequacy. However, there is an important theoretical reason for not doing so. It is clearly desirable on intuitive grounds that any notion of adequacy satisfy the following condition or axiom which we call monotonicity:

We say that an adequacy criterion is monotonic if whenever T is adequate for a program P and $T \subseteq T'$, then T' is also adequate for P.

Monotonicity simply requires that additional test data should never transform an adequate test set into an inadequate one. But if we made correctness on the test set part of a criterion for adequacy, the addition to an adequate test set of a new input on which the program produces an incorrect output would transform the adequate test set to an inadequate one! Of course additional test data may well transform a non-adequate test set into one which is adequate by our definition. All that is necessary is that the additional data suffice to distinguish the program from "simpler" programs in which it is not embedded.

Needless to say, the fact that a program is correct on a set of test data which fulfills our adequacy criterion does not suffice to guarantee that the program is really correct. Our proposed notion of test data adequacy is defined relative to a given program, and depends on the specification only to the extent that we expect the program be correct for every element of the test set T before applying the notion. This lack of dependence on the specification is a property of all program-based testing strategies and adequacy notions. Our notion certifies that the program as written has been adequately tested, but does not tell us how well the program meets the specification except at the selected test points. Since "missing logic," i.e. a failure to fulfill some portion of the specification, is a real and frequent problem [11,12], we feel that in practice,

be invoked unless it is believed that the program agrees with its specification on the test set. Hence, it might be thought appropriate to include correctness on the test set as part of the definition of adequacy. However, there is an important theoretical reason for not doing so. It is clearly desirable on intuitive grounds that any notion of adequacy satisfy the following condition or axiom which we call monotonicity:

We say that an adequacy criterion is monotonic if whenever T is adequate for a program P and $T \subseteq T'$, then T' is also adequate for P.

Monotonicity simply requires that additional test data should never transform an adequate test set into an inadequate one. But if we made correctness on the test set part of a criterion for adequacy, the addition to an adequate test set of a new input on which the program produces an incorrect output would transform the adequate test set to an inadequate one! Of course additional test data may well transform a non-adequate test set into one which is adequate by our definition. All that is necessary is that the additional data suffice to distinguish the program from "simpler" programs in which it is not embedded.

Needless to say, the fact that a program is correct on a set of test data which fulfills our adequacy criterion does not suffice to guarantee that the program is really correct. Our proposed notion of test data adequacy is defined relative to a given program, and depends on the specification only to the extent that we expect the program be correct for every element of the test set T before applying the notion. This lack of dependence on the specification is a property of all program-based testing strategies and adequacy notions. Our notion certifies that the program as written has been adequately tested, but does not tell us how well the program meets the specification except at the selected test points. Since "missing logic," i.e. a failure to fulfill some portion of the specification, is a real and frequent problem [11,12], we feel that in practice,

it is essential that any adequacy notion based on ideas like those being discussed be used in conjunction with a test data selection criterion which is not entirely program-based. Clearly program-based selection criteria are poor at exposing this kind of error. Instead we recommend that an error-based test data selection criterion, like those discussed in [1, 4, 7, 8, 13, 14, 16], which draws heavily on the specification, be used in conjunction with criteria based on our ideas.

Comparison with Other Adequacy Criteria

We begin by showing that the present notion essentially subsumes branch coverage. This latter criterion regards a test set T as adequate if running the program on all the elements of T results in each branch of the program graph being traversed at least once. Since branch coverage is a widely used adequacy criterion, this comparison is of substantial interest.

Thus, assume the program P contains the program body:

```
if PRED then Q
        else R
```

Let P be reduced to P′ by replacing the above program body by R. Now, if in the program P, the branch Q is never taken when inputs are chosen from T, then $P(t) = P'(t)$ for all $t \in T$. By definition, our measure of program size is such that $|P'| \leq |P|$. Finally, since P reduces to P′, if P is embedded in P′, then P must be (non-trivially) embedded in itself. Let us call a program P self-embedded if there is a program Q such that P reduces to Q, $Q \equiv P$, and Q is not identical to P. Thus, we have proved:

THEOREM 2. Let P be a program which is not self-embedded, and let T be an adequate test set for P. Then T satisfies the branch coverage criterion for P.

Mutation analysis is an adequacy notion which has enjoyed a great deal of interest [1, 4]. Given a program P and a test set T such that P is correct on every member of T, a set of alternative programs known as mutants of P must be produced. Each mutant P_i is formed by modifying a single statement of P in some predefined way. The resulting P_i's each have the property that $|P_i| \leq |P|$. Each mutant is then run on every element of T, and T is said to be mutation adequate for P provided that for every inequivalent mutant P_i of P, there is a $t \in T$ such that $P_i(t) \neq P(t)$. Clearly, mutation adequacy can be thought of as an approximation to our original notion of size-adequacy, in the sense that instead of requiring that a test set be sufficient to distinguish P from all inequivalent shorter programs, it need only distinguish P from a predefined subset of these programs. This restriction has the desirable effect of solving the problem discussed in Theorem 1, that nonoptimal programs have only trivial adequate test sets. We readily obtain:

THEOREM 3. Let P be a program which is not self-embedded, and let T be an adequate test set for P. Then T distinguishes P from each of its mutants.

Critical Points

Our next concern is to consider points determined by a given program which must be included in every adequate test set for that program. To this end, we introduce the notion of a critical point. We say that an element c is critical for a program P if there is a program P′ in which P is not embedded such that $|P'| \leq |P|$, $P'(c) \neq P(c)$, and for all $x \neq c$, $P'(x) = P(x)$.

Intuitively, if a program can be wrong at exactly one point, that point is critical. It follows immediately from the definitions that:

Mutation analysis is an adequacy notion which has enjoyed a great deal of interest [1, 4]. Given a program P and a test set T such that P is correct on every member of T, a set of alternative programs known as mutants of P must be produced. Each mutant P_i is formed by modifying a single statement of P in some predefined way. The resulting P_i's each have the property that $|P_i| \leq |P|$. Each mutant is then run on every element of T, and T is said to be mutation adequate for P provided that for every inequivalent mutant P_i of P, there is a $t \in T$ such that $P_i(t) \neq P(t)$. Clearly, mutation adequacy can be thought of as an approximation to our original notion of size-adequacy, in the sense that instead of requiring that a test set be sufficient to distinguish P from all inequivalent shorter programs, it need only distinguish P from a predefined subset of these programs. This restriction has the desirable effect of solving the problem discussed in Theorem 1, that nonoptimal programs have only trivial adequate test sets. We readily obtain:

THEOREM 3. Let P be a program which is not self-embedded, and let T be an adequate test set for P. Then T distinguishes P from each of its mutants.

Critical Points

Our next concern is to consider points determined by a given program which must be included in every adequate test set for that program. To this end, we introduce the notion of a critical point. We say that an element c is critical for a program P if there is a program P' in which P is not embedded such that $|P'| \leq |P|$, $P'(c) \neq P(c)$, and for all $x \neq c$, $P'(x) = P(x)$.

Intuitively, if a program can be wrong at exactly one point, that point is critical. It follows immediately from the definitions that:

THEOREM 4. If T is a adequate test set for a program P and c is a critical point for P, then c is in T.

Clearly, an adequate test set for P may contain points which are not critical for P. This follows immediately from the fact that our adequacy criterion is monotonic. We shall see in the next section, however, that even when restrictions are placed on the size of an adequate test set, it may still contain non-critical points.

The manner in which the notion of critical point arises naturally in the context of our notion of test adequacy is particularly striking when one considers the types of points which turn out to be critical.

Example 1

```
P:  input x
    if x ≤ b then Q
             else R
    output y
```

For this program, b will ordinarily be a critical point, since we can construct the following program which is of equal size and behaves exactly like P except possibly at the point b,

```
P′: input x
    if x < b then Q
             else R
    output y
```

In fact $P(b) \neq P'(b)$ unless Q and R produce the same outputs on input b. Of course

our definition will not apply if P is embedded in P′. But in this example this will only happen if P is self-embedded which we can rule out as a pathological case.

It is interesting to note that since P can be reduced to the following program:

```
input x
   R
output y
```

if P is equivalent to this program, then b is not a critical point for P. In that case, Q and R must produce the same outputs on all inputs $x \leq b$, and so the test for $x \leq b$, is in fact useless. In such a case, the non-embeddedness condition in our definition of critical point is consistent with our intuition about the non-critical nature of the point b in program P.

P will also have c=b+1 as a critical point for which the following program will have the required properties:

```
P′′: input x
     if x ≤ c then Q
              else R
     output y
```

In each case above, the critical point is a boundary value, and the programs P′ and P′′ involve the shifting of the boundary by 1. Here, P′ and P′′ contain what is commonly called an off-by-one error.

our definition will not apply if P is embedded in P′. But in this example this will only happen if P is self-embedded which we can rule out as a pathological case.

It is interesting to note that since P can be reduced to the following program:

```
input x
   R
output y
```

if P is equivalent to this program, then b is not a critical point for P. In that case, Q and R must produce the same outputs on all inputs $x \leq b$, and so the test for $x \leq b$, is in fact useless. In such a case, the non-embeddedness condition in our definition of critical point is consistent with our intuition about the non-critical nature of the point b in program P.

P will also have c=b+1 as a critical point for which the following program will have the required properties:

```
P′′: input x
     if x ≤ c then Q
              else R
     output y
```

In each case above, the critical point is a boundary value, and the programs P′ and P′′ involve the shifting of the boundary by 1. Here, P′ and P′′ contain what is commonly called an off-by-one error.

Example 2

```
P:  input x
    if x = b then Q
             else R
    output y
```

For this program, b is a critical point provided that $Q(b) \neq R(b)$, since the program:

```
    input x
       R
    output y
```

has size no larger than $|P|$ and produces the same output as P for every element other that b. This corresponds to the case in which the programmer has "forgotten" to test for the special case x=b. b can also be seen to be a critical point by considering the following program:

```
P': input x
    if x = b then Q'
             else R
    output y
```

where Q' is a program body such that $Q(b) \neq Q'(b)$ and $|Q'| \leq |Q|$. Intuitively, this corresponds to the case in which the program does check for the proper special value but the wrong action is taken.

Notice that a value which is a critical point does not necessarily occur explicitly in one of the predicates of the program. Rather, a critical point is

an input value which results in a (possibly computed) value that produces "boundary" behavior for one of the predicates.

What these examples indicate is that in order for a test set to be adequate by our definition, boundary points and special values must be included in the test set. Since it has long been recognized by testing practitioners that such points should be included in test sets, we consider this to be strong corroborating evidence that our proposed notion of adequacy is a reasonable one. In a later section, we shall show that it is not sufficient to restrict attention solely to these points. In fact in order for a test set to be adequate in general, some "central" cases must also be included.

Bounds on the Size of an Adequate Test Set

We now show that for every program P, there exists a set which is an adequate test set for P. Let P be a given program. For each program Q in which P is not embedded for which $|Q| \leq |P|$, let $t = t_Q$ be an element such that $Q(t) \neq P(t)$ and let $T = \{t_Q\}$ for all such programs Q. By our assumptions, T is a finite set, and it is clearly adequate for P. Note that the number of elements in T is less than or equal to the number of elements in the set $\{Q: |Q| \leq |P|\}$. But of course this upper bound is grossly impractical; we will return to our discussion of upper bounds later in this section.

We will now show how our definition leads readily to a lower bound on the number of elements in an adequate test set for a given program.

THEOREM 5. Let T be an adequate test set of size n for the program P, and let P be defined on more than n elements. Then, $n > |P|$.

Proof: Let $T = \{t_1, t_2, \ldots, t_n\}$ and $P(t_i) = b_i$, $i=1,2,\ldots,n$. Let Q be the following "table lookup" program:

an input value which results in a (possibly computed) value that produces "boundary" behavior for one of the predicates.

What these examples indicate is that in order for a test set to be adequate by our definition, boundary points and special values must be included in the test set. Since it has long been recognized by testing practitioners that such points should be included in test sets, we consider this to be strong corroborating evidence that our proposed notion of adequacy is a reasonable one. In a later section, we shall show that it is not sufficient to restrict attention solely to these points. In fact in order for a test set to be adequate in general, some "central" cases must also be included.

Bounds on the Size of an Adequate Test Set

We now show that for every program P, there exists a set which is an adequate test set for P. Let P be a given program. For each program Q in which P is not embedded for which $|Q| \leq |P|$, let $t = t_Q$ be an element such that $Q(t) \neq P(t)$ and let $T = \{t_Q\}$ for all such programs Q. By our assumptions, T is a finite set, and it is clearly adequate for P. Note that the number of elements in T is less than or equal to the number of elements in the set $\{Q: |Q| \leq |P|\}$. But of course this upper bound is grossly impractical; we will return to our discussion of upper bounds later in this section.

We will now show how our definition leads readily to a lower bound on the number of elements in an adequate test set for a given program.

THEOREM 5. Let T be an adequate test set of size n for the program P, and let P be defined on more than n elements. Then, $n > |P|$.

Proof: Let $T = \{t_1, t_2, \ldots, t_n\}$ and $P(t_i) = b_i$, $i=1,2,\ldots,n$. Let Q be the following "table lookup" program:

```
input x
if x = t1 then y ← b1 else
if x = t2 then y ← b2 else
... ... ... ... ...
... ... ... ... ...
if x = tn then y ← bn
output y
```

Clearly $Q(t) = P(t)$ for each $t \in T$. Since Q is only defined on T and P is defined on more than n elements P cannot be embedded in Q. Since T is adequate for P it follows that $|Q| > |P|$. The desired result follows since $|Q| = \max(n,n) = n$. ●

It is true that this exact lower bound depends directly on our assumptions on how program size is measured. Thus, for example, if we consider assignment statements to be of length 1, input/output statements as of length 0, and statements of the form:

if PRED then Q else R

(where Q and R are program bodies) to be of length $(1+|Q|+|R|)$, then we obtain the bound $n > |P|/2 - 1$. Obviously other variations are possible, but the important point is that we have a established a lower bound on the size of an adequate test set which, under various reasonable measures of program size, is directly proportional to the length of the program being tested.

We say that the set T is minimally adequate for program P if T is adequate for P but no proper subset of T is adequate for P. For any set of test data which is adequate for a given program without being minimally adequate, there is a proper subset which is still adequate. Since, as we saw, every program possesses

an adequate test set, iteration of this process leads to a minimally adequate set of test data in a finite number of (not necessarily constructive) steps.

In the last section, we pointed out that although every critical point must be in an adequate test set, an arbitrary adequate test set may well contain non-critical points. An obvious question is whether or not a minimally adequate test set for a program is exactly the set of critical points for the program. The next example shows that this is not the case.

Example 3

```
P:  input x
    if x = 0 then y ← 0
             else y ← 1
    output y
```

The program:

```
P': input x
    y ← 1
    output y
```

shows that 0 is a critical point for P. Furthermore, if a program Q agrees with P on all points except for a single non-zero point, then $|Q| > 2 = |P|$. Hence, it follows that 0 is the only critical point for P. However, {0} is not an adequate test set for P since the program:

```
P'': input x
     if x = 0 then y ← 0
              else y ← 2
     output y
```

an adequate test set, iteration of this process leads to a minimally adequate set of test data in a finite number of (not necessarily constructive) steps.

In the last section, we pointed out that although every critical point must be in an adequate test set, an arbitrary adequate test set may well contain non-critical points. An obvious question is whether or not a minimally adequate test set for a program is exactly the set of critical points for the program. The next example shows that this is not the case.

Example 3

```
P:  input x
    if x = 0 then y ← 0
             else y ← 1
    output y
```

The program:

```
P': input x
    y ← 1
    output y
```

shows that 0 is a critical point for P. Furthermore, if a program Q agrees with P on all points except for a single non-zero point, then $|Q| > 2 = |P|$. Hence, it follows that 0 is the only critical point for P. However, {0} is not an adequate test set for P since the program:

```
P'': input x
     if x = 0 then y ← 0
              else y ← 2
     output y
```

is correct on 0, P is not embedded in P'', but $|P''| = |P| = 2$.

Thus, although it is necessary to test a program at critical points such as boundary points and special values, it is not sufficient to restrict attention to these points. The next examples provide further illustrations of this point.

Example 4

```
P:  input x
    if x > 10 then y ← 1
              else y ← x+3
    output y
```

The critical points for this program are 10 and 11. They do not, however, by themselves form an adequate test set, even though both branches of the condition are traversed. It is easy to demonstrate that in order for a test set to be adequate for this program, it must include at least one input less than 10 and one input greater than 11. Thus our notion requires in such a case that each branch be traversed at least twice; once at each side of the boundary, and at one other "central" point on each side of the boundary. A similar situation holds for the other inequality predicates.

Example 5

```
P:  input x
    if x = 10 then y ← 1
              else y ← x+3
    output y
```

For this case 10 is the only critical point. An adequate test set for this program must contain, in addition to 10, at least one input greater than 10 and

at least one input less than 10. A similar situation holds for the predicate "$\neq$".

Again, we are encouraged by the fact that this is consistent with well-established testing practice. Surely, in addition to testing the special cases, some "central" cases must also be included in a test set.

We next consider properties of programs for which a given test set is minimally adequate.

Let $T = \{t_1, t_2, \ldots, t_n\}$ be minimally adequate for program P. Let $T_i = T - \{t_i\}$, $i = 1,2,\ldots,n$. Since T_i can not be adequate for P, there must be a program P_i (in which P is not embedded) such that $P_i(x) = P(x)$ for all x in T_i, $P_i(t_i) \neq P(t_i)$ and $|P_i| \leq |P|$. Let $P(t_i) = b_i$, $i = 1,2,\ldots,n$, and let Q_i be the following program:

```
input x
if x = t_i then y ← b_i
           else P_i
output y
```

Then,

$$|Q_i| = |P_i| + 1.$$

Moreover, $Q_i(x) = P(x)$ for all x in T. Can P be embedded in Q_i? Since P is not embedded in P_i, this can only happen if P always outputs the value b_i. To avoid this case, let us assume that P outputs at least two distinct values. Then, P cannot be embedded in Q_i, and, since T is adequate for P, we must have that either $Q_i \equiv P$ or $|Q_i| > |P|$. In the latter case we can write:

$$|Q_i| = 1 + |P_i| \leq 1 + |P| < 1 + |Q_i|,$$

which implies that $|P| = |P_i|$. We have thus proved:

at least one input less than 10. A similar situation holds for the predicate "$\neq$".

Again, we are encouraged by the fact that this is consistent with well-established testing practice. Surely, in addition to testing the special cases, some "central" cases must also be included in a test set.

We next consider properties of programs for which a given test set is minimally adequate.

Let $T = \{t_1, t_2, \ldots, t_n\}$ be minimally adequate for program P. Let $T_i = T - \{t_i\}$, $i = 1,2,\ldots,n$. Since T_i can not be adequate for P, there must be a program P_i (in which P is not embedded) such that $P_i(x) = P(x)$ for all x in T_i, $P_i(t_i) \neq P(t_i)$ and $|P_i| \leq |P|$. Let $P(t_i) = b_i$, $i = 1,2,\ldots,n$, and let Q_i be the following program:

```
input x
if x = t_i then y ← b_i
           else P_i
output y
```

Then,

$$|Q_i| = |P_i| + 1.$$

Moreover, $Q_i(x) = P(x)$ for all x in T. Can P be embedded in Q_i? Since P is not embedded in P_i, this can only happen if P always outputs the value b_i. To avoid this case, let us assume that P outputs at least two distinct values. Then, P cannot be embedded in Q_i, and, since T is adequate for P, we must have that either $Q_i \equiv P$ or $|Q_i| > |P|$. In the latter case we can write:

$$|Q_i| = 1 + |P_i| \leq 1 + |P| < 1 + |Q_i|,$$

which implies that $|P| = |P_i|$. We have thus proved:

THEOREM 6 (Dichotomy Theorem). Let $T = \{t_1, t_2, \ldots, t_n\}$ be a minimally adequate test set for a program P which outputs at least two distinct values. Let P_i be as above, $i = 1,2,\ldots,n$. Then, for each i, either $P(x) = P_i(x)$ for all $x \neq t_i$ or $|P| = |P_i|$.

This result can be restated in terms of critical points. In that form we have:

THEOREM 6. Let $T = \{t_1, t_2, \ldots, t_n\}$ be a minimally adequate test set for a program P which outputs at least two distinct values, let P_i be as above, $i = 1,2,\ldots,n$, and let t_i be a non-critical point for P. Then $|P| = |P_i|$.

Since we have shown above that a minimally adequate test set may contain non-critical points, we next consider the problem of obtaining a bound on the number of non-critical points in a minimally adequate test set. The following result follows immediately from the Theorem 6 and the definition of critical point:

COROLLARY The number of non-critical points in a minimally adequate test set for a program P which outputs at least two distinct values is no greater than the number of points in the domain for which there exist a program which is the same length as P and which differs from P only at the one designated point.

Conclusions

We have introduced a formal notion of test data adequacy which subsumes such popular adequacy notions as branch coverage and mutation analysis. In addition, our notion requires that in order for a test set to be adequate, it must include critical points of the program. These points turn out to be, in general, exactly the types of elements of the domain which are commonly recognized as being

particularly error prone, namely boundary cases and special values. We show, however, that it is not sufficient to focus attention solely on these points. Thus an adequate test set must include "central" cases as well as the critical points.

We have shown too, that there is an easily computed lower bound on the size of an adequate test set in terms of the size of the program, as well as an upper bound on the size of a minimally adequate test set.

Acknowledgements

We are pleased to acknowledge helpful conversations with James Collofello, Tom Ostrand, and Scott Woodfield.

REFERENCES

[1] T.A. Budd, R.J. Lipton, R.A. DeMillo, and F.G. Sayward, "Mutation Analysis," Dept of Computer Science Res Rpt 155, Yale University, New Haven, Ct, April 1979.

[2] N. Chapin, "A Measure of Software Complexity", Proc. of the 1979 National Computer Conference, New York, pp. 995-1002.

[3] M. Davis and E.J. Weyuker, "Pseudo-Oracles for Non-testable Programs", Proc. ACM National Conference, Los Angeles, November 1981.

[4] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, 11(4), April 1978, pp. 34-41.

[5] J.L. Elshoff and M. Marcotty, "On the Use of the Cyclomatic Number to Measure Program Complexity," SIGPLAN Notices, Vol.13, No.12, Dec 1978, pp.29-40.

[6] W.J. Hansen, "Measurement of Program Complexity by the Pair (Cyclomatic Number,Operator Count)," SIGPLAN Notices, Vol.13, No.3, March 1978, pp.29-33.

[7] W.E. Howden, "Algebraic Program Testing," Acta Informatica, Vol.10, 1978, pp.53-66.

[8] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets", IEEE Trans. Software Eng., Vol. SE-8, July 1982, pp.371-379.

[9] T.J. McCabe, "A Complexity Measure," IEEE Trans. Software Eng., Vol.sIE-2, No.4, Dec 1976, pp.308-320.

particularly error prone, namely boundary cases and special values. We show, however, that it is not sufficient to focus attention solely on these points. Thus an adequate test set must include "central" cases as well as the critical points.

We have shown too, that there is an easily computed lower bound on the size of an adequate test set in terms of the size of the program, as well as an upper bound on the size of a minimally adequate test set.

Acknowledgements

We are pleased to acknowledge helpful conversations with James Collofello, Tom Ostrand, and Scott Woodfield.

REFERENCES

[1] T.A. Budd, R.J. Lipton, R.A. DeMillo, and F.G. Sayward, "Mutation Analysis," Dept of Computer Science Res Rpt 155, Yale University, New Haven, Ct, April 1979.

[2] N. Chapin, "A Measure of Software Complexity", Proc. of the 1979 National Computer Conference, New York, pp. 995-1002.

[3] M. Davis and E.J. Weyuker, "Pseudo-Oracles for Non-testable Programs", Proc. ACM National Conference, Los Angeles, November 1981.

[4] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on Test Data Selection: Help for the Practicing Programmer", Computer, 11(4), April 1978, pp. 34-41.

[5] J.L. Elshoff and M. Marcotty, "On the Use of the Cyclomatic Number to Measure Program Complexity," SIGPLAN Notices, Vol.13, No.12, Dec 1978, pp.29-40.

[6] W.J. Hansen, "Measurement of Program Complexity by the Pair (Cyclomatic Number,Operator Count)," SIGPLAN Notices, Vol.13, No.3, March 1978, pp.29-33.

[7] W.E. Howden, "Algebraic Program Testing," Acta Informatica, Vol.10, 1978, pp.53-66.

[8] W.E. Howden, "Weak Mutation Testing and Completeness of Test Sets", IEEE Trans. Software Eng., Vol. SE-8, July 1982, pp.371-379.

[9] T.J. McCabe, "A Complexity Measure," IEEE Trans. Software Eng., Vol.sIE-2, No.4, Dec 1976, pp.308-320.

[10] G.J. Myers, "An Extension to the Cyclomatic Measure of Program Complexity," SIGPLAN Notices, Vol.12, No.10, Oct 1977, pp.61-64.

[11] T.J. Ostrand and E.J. Weyuker, "Collecting and Categorizing Software Error Data in an Industrial Environment," Dept of Computer Science Technical Rpt 047, Courant Institute of Mathematical Sciences, New York University, New York, August 1982.

[12] T.A. Thayer, M. Lipow, and E.C. Nelson, Software Reliability, A Study of Large Project Reality, North Holland Publishing Co., New York, 1978.

[13] E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," IEEE Trans. Software Eng., Vol.slE-6, May 1980, pp.236-246.

[14] E.J. Weyuker, "An Error-Based Testing Strategy," Dept of Computer Science Technical Rpt 027, Courant Institute of Mathematical Sciences, New York University, New York, January 1981.

[15] E.J. Weyuker, "On Testing Non-testable Programs," The Computer Journal, Vol.25, No.4, 1982.

[16] L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing," IEEE Trans. Software Eng., Vol. SE-6, May 1980, pp.247-257.

[17] M.R. Woodward, M.A. Hennell, and D. Hedley, "A Measure of Control Flow Complexity in Program Text," IEEE Trans. Software Eng., Vol.SE-5, No.1, Jan 1979, pp.45-50.

NYU COMPSCI TR-50 c.1
Davis, Martin D
A formal notion of program-
based test data adequacy.